



Playwright

Reliable end-to-end testing for modern web apps

Last updated: 23 May 2023

Contents

About Playwright	5
Why Playwright?	5
Multiple language support.....	5
Support for all browser families	5
Fast and reliable execution	6
Powerful automation capabilities.....	6
JavaScript Playwright installation.....	7
Install Node.js and the Node Package Manager (npm)	7
Install Playwright (choose one of the two options).....	7
Option 1: Automated Playwright installation	7
Option 2: Manual Playwright installation	7
Install an Integrated Development Environment (IDE) for JavaScript.....	7
Java Playwright installation	8
Install Java	8
Install Maven.....	8
Playwright with Apache Maven	8
Install an Integrated Development Environment (IDE) for Java	8
JavaScript Synchronous and asynchronous test executions	9
JavaScript Configuration file.....	9
Reporters	10
Tracing.....	11
UI Mode	12
JavaScript Test Hooks and Fixtures	14
JavaScript Command line patterns.....	15
Core concepts of Playwright	16
Browser	16
Browser contexts	16
Pages and frames	16
Selectors.....	16
Inspect Selectors	17
Locators.....	18
Built-in locators.....	18
Playwright Inspector	19
Open the Playwright Inspector	19

Java and JavaScript Command Line Tools	20
Record scripts automatically.....	20
Preserve and restore authenticated state from command line	20
Install browsers.....	20
Open a page with browsers and emulation options.....	21
Open Chromium:.....	21
Open WebKit:.....	21
Open emulate an Apple iPhone 13.	21
Open emulate colour scheme and viewport (screen) size	21
Open emulate geolocation, language and timezone	21
Take Screenshot.....	21
Generate PDF	21
Auto Waiting	22
Inputs	23
Text input (“fill”)	23
Checkboxes and radio buttons (“check” and “uncheck”).....	23
Select options (“selectOption”)	23
Mouse click (“click” and “dblclick”)	23
Type characters (“type”).....	23
Keys and shortcuts (“press”).....	24
Java and JavaScript Assertions	24
Java Asserption examples.....	24
JavaScript Assertion examples (just some of the same Assertions as in Java)	24
Java and JavaScript Reuse authentication states programmatically	25
Save the authentication state programmatically	25
Create a new context with the saved storage state	25
Emulation.....	25
Browser Flags and configuration settings (Options).....	25
Chromium Flags	26
Mozilla Firefox configuration settings	26
JavaScript Data-Driven Tests	27
JavaScript Tags.....	28
Example use of tags	28
Filtering with grep.....	28
Command line tag filtering examples:	28

Configuration file tag filtering example	28
Playwright on Microsoft .net and Azure DevOps.....	29
Playwright installation without an IDE (NUnit).....	29
Playwright installation in Visual Studio (NUnit without SpecFlow)	29
Playwright installation in Visual Studio (NUnit with SpecFlow).....	29
Additional installations for Azure DevOps	30
Install Browsers on Microsoft .net (in PowerShell)	31
Running tests on Microsoft .net	31
Visual Studio “.runsettings” file	31

Disclaimer

Some content of this document has been sourced from the official Playwright web site:
<https://playwright.dev/>.

The Playwright team really did an excellent job in documenting the Playwright features and providing useful examples.

About Playwright

Playwright is a free open source library for browser automation developed and maintained by Microsoft. Like Selenium, Playwright offers multiple language bindings, including Java.

The JavaScript/TypeScript version of Playwright has its own test runner, called Playwright Test. This document describes both the JavaScript version of Playwright, and the Java library use of Playwright. The Java library version can be used with any Java (unit) test framework, such as JUnit or TestNG.

This document also describes Playwright on the Microsoft .net platform. Playwright can be used with any of the Microsoft .net programming languages, such as C#.

Playwright is led by the same team that originally built Puppeteer at Google. It therefore builds on the strength of Puppeteer, particularly on browser context flexibility and network manipulation capabilities, including handling and modifying of requests (stubbing and mocking).

Unlike Cypress, Playwright does not run inside a browser and does therefore not suffer from the same browser security limitations as Cypress, such as cross-site scripting restrictions (for example when using external authentication).

Why Playwright?

Playwright enables fast, reliable and capable testing and automation across all modern browsers.

Multiple language support

Comparable to Selenium, the Playwright API is available in multiple languages:

- JavaScript and TypeScript
- Python
- Java
- Microsoft .net

Support for all browser families

- Playwright runs on Chromium, Firefox and WebKit. Playwright has full API coverage for all modern browsers, including Google Chrome and Microsoft Edge (with Chromium), Apple Safari (with WebKit), and Mozilla Firefox. Headless execution is supported for all browsers on all platforms.
- Cross-platform WebKit testing. With Playwright, you can test how your app behaves in Apple Safari with WebKit builds for Windows, Linux and macOS. In other words: You can reproduce Apple Safari on non-Apple operating systems! Tests can run locally and in Continuous Integration.
- Tests for mobile. Use device emulation to test your responsive web apps in mobile web browsers.
- Headless and headed. Playwright supports headless (without browser UI) and headed (with browser UI) modes for all browsers and all platforms. Headed is great for debugging, and headless is faster and suited for Continuous Integration and cloud executions.

Fast and reliable execution

- Auto-wait API's. Playwright interactions auto-wait for elements to be ready. This improves reliability and simplifies test authoring.
- Timeout-free automation. Playwright receives browser signals, such as network requests, page navigations, and page load events to eliminate the need for sleep timeouts that cause flakiness. This is a major advantage over Selenium.
- Fast isolation with browser contexts. Reuse a single browser instance for multiple isolated execution environments with browser contexts.
- Resilient element selectors. Playwright can rely on user-facing strings, like text content and accessibility labels, to select elements. These strings are more resilient than selectors that are tightly-coupled to the DOM structure.
- React and Vue selectors. Playwright has customised element selectors for the React and Vue web frameworks. It also supports Svelte.
- All selector engines except for XPath pierce shadow DOM by default. This is a major breakthrough advantage over Selenium and crucially important for modern web apps.

Powerful automation capabilities

- Multiple domains, pages and frames. Playwright is an out-of-process automation driver that is not limited by the scope of in-page JavaScript execution. It can automate scenarios with multiple pages.
- Powerful network control. Playwright introduces context-wide network interception to stub and mock network requests.
- Modern web features. Playwright supports web components through shadow-piercing selectors, geolocation, permissions, web workers, and other modern web API's.
- Capabilities to cover all scenarios. Support for file downloads and uploads, out-of-process iframes, native input events, and even dark mode.
- UI Mode. Explore, run and debug tests in UI Mode with a time travel experience complete with watch mode. All test files are loaded into the testing sidebar where you can expand each file and describe block to individually run, view, watch and debug each test. See a full trace of your tests and hover back and forward over each action to see what was happening during each step and pop out the DOM snapshot to a separate window for a better debugging experience.

JavaScript Playwright installation

Install Node.js and the Node Package Manager (npm)

Before you can install and use Playwright, you need to install Node.

Please follow an installation guide for your operating system and version.

You can check that Node has been successfully installed with this command:

```
node -v
```

When you installed Node, you also automatically installed the “npm” Command Line Interface (CLI), which is the package manager for Node. You can check it with:

```
npm -v
```

Install Playwright (choose one of the two options)

Option 1: Automated Playwright installation

This will create a Playwright Test configuration file, optionally add examples, a GitHub Action workflow, and a first test:

```
npm init playwright@latest
```

Option 2: Manual Playwright installation

If you want to use Playwright with the JavaScript test-runner, then you need to run the following two commands (the second command installs the supported browsers):

```
npm i -D @playwright/test
```

```
npm run playwright install --with-deps
```

Alternatively, if you just want to use the core of Playwright without the JavaScript test-runner, then you can use this command:

```
npm i -D playwright
```

Install an Integrated Development Environment (IDE) for JavaScript

An Integrated Development Environment (IDE), such as Microsoft Visual Studio Code, is recommended for development:

<https://code.visualstudio.com/download>

There is an official Microsoft Visual Studio Code extension from Microsoft called “Playwright Test for VSCode”, which is highly recommended:

<https://marketplace.visualstudio.com/items?itemName=ms-playwright.playwright>

Java Playwright installation

Install Java

Java JDK version 8 (Standard Edition) or higher (Java long-term support versions are recommended) can be installed either from Oracle

<https://www.oracle.com/java/technologies/javase-downloads.html> ,

or alternatively a version of the Java Open JDK, for example from:

<https://openjdk.java.net/>

Install Maven

The latest stable version of Apache Maven version 3 or higher can be installed from:

<https://maven.apache.org/download.cgi>

Playwright with Apache Maven

Playwright is distributed as a set of Apache Maven modules. The easiest way to use it is to add one dependency to your project's "pom.xml":

```
<dependency>
  <groupId>com.microsoft.playwright</groupId>
  <artifactId>playwright</artifactId>
  <version>1.32.0</version>
</dependency>
```

Install an Integrated Development Environment (IDE) for Java

An Integrated Development Environment (IDE), such as IntelliJ IDEA Community Edition, is recommended for development:

<https://www.jetbrains.com/idea/download/>

JavaScript Synchronous and asynchronous test executions

One of the key differences between the JavaScript and the Java versions of Playwright is that the JavaScript version works asynchronous, while the Java version works synchronous. This means that by default in JavaScript, a new command will not wait for the previous command to be completed before it starts. This can lead to errors and unexpected results.

In JavaScript, many commands must therefore be instructed to wait for their own completion before the script can continue. The “`await`” keyword will make the following function wait for a Promise, before the script can continue.

In JavaScript, it is therefore strongly encouraged to use “`async`” and “`await`”, as in this example:

```
const { test, expect } = require('@playwright/test');
test('basic test', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  const title = page.locator('.navbar__inner .navbar__title');
  await expect(title).toHaveText('Playwright');
});
```

JavaScript Configuration file

A configuration file with the file name “`playwright.config.js`” in the root folder of the project can be used to define the desired test execution by default. However, if you don’t use a configuration file, you can also set the test execution (such as reporters and projects) via command line (CLI) options.

The following example configuration runs every test in desktop Chromium, Firefox, WebKit, Apple iOS (iPhone 13 Pro), and Google Android (Pixel 5) by creating a “project” for each browser configuration. It also specifies two retries on Continuous Integration (CI) only.

It is also possible to set “project” dependencies, so that a particular (pre-requisite) “project” or even a group of projects have to run before a (follow-up) “project” gets executed.

The following example uses a “`baseURL`” so that URL’s in tests can omit this part of the full URL and just use the part after the “`baseURL`” instead (for example just “`/myFolderName/myFileName`”), only take screenshots on failure, not record videos of the test executions, and only record tracing information on the first retry (after a failure).

```

const { devices } = require('@playwright/test');
const config = {
  /* Fail the build on CI if test.only is left in the source code */
  forbidOnly: !!process.env.CI,
  /* Retry on CI only */
  retries: process.env.CI ? 2 : 0,
  /* GitHub and HTML reporters on CI, HTML only locally */
  reporter: process.env.CI ? [['github'],['html']] : [['html',{open:'never'}]],
  use: {
    baseURL: 'https://www.mywebsite.com',
    screenshot: 'only-on-failure',
    video: 'off',
    trace: 'on-first-retry'
  },
  projects: [
    {
      name: 'Desktop Chrome',
      browserName: 'chromium',
      viewport: { width: 1280, height: 720 }
    },
    {
      name: 'Desktop Firefox',
      browserName: 'firefox',
      viewport: { width: 1280, height: 720 }
    },
    {
      name: 'Desktop Safari',
      browserName: 'webkit',
      viewport: { width: 1280, height: 720 }
    },
    {
      name: 'Apple iOS',
      use: { ...devices['iPhone 13 Pro'] },
    },
    {
      name: 'Google Android',
      use: { ...devices['Pixel 5'] },
    }
  ],
};
module.exports = config;

```

Reporters

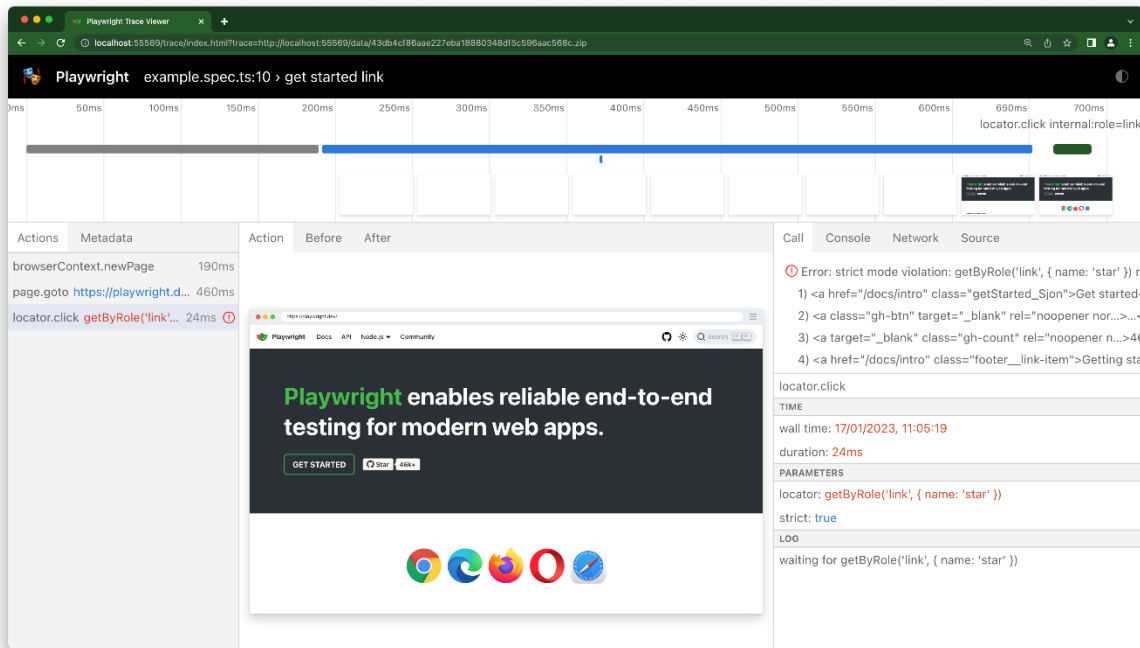
Playwright provides multiple reporters. The example local configuration uses the HTML report using the default folder “playwright-report” without automatically opening the report after completion. It is also possible to use multiple reporters at the same time.

Tracing

Recording videos for debugging is not recommended in Playwright. Instead of recording videos, tracing should be used. Tracing creates a “trace.zip” file per test that contains Playwright actions, Playwright events, DOM snapshots, screenshots, network log, and console log.

The “trace.zip” files can be viewed with the “Trace Viewer”, which is a GUI tool that is bundled with Playwright. “trace.zip” files get stored in the “test-results” folder. They can be viewed with:

```
npm run playwright show-trace test-results/NameOfTest/trace.zip
```



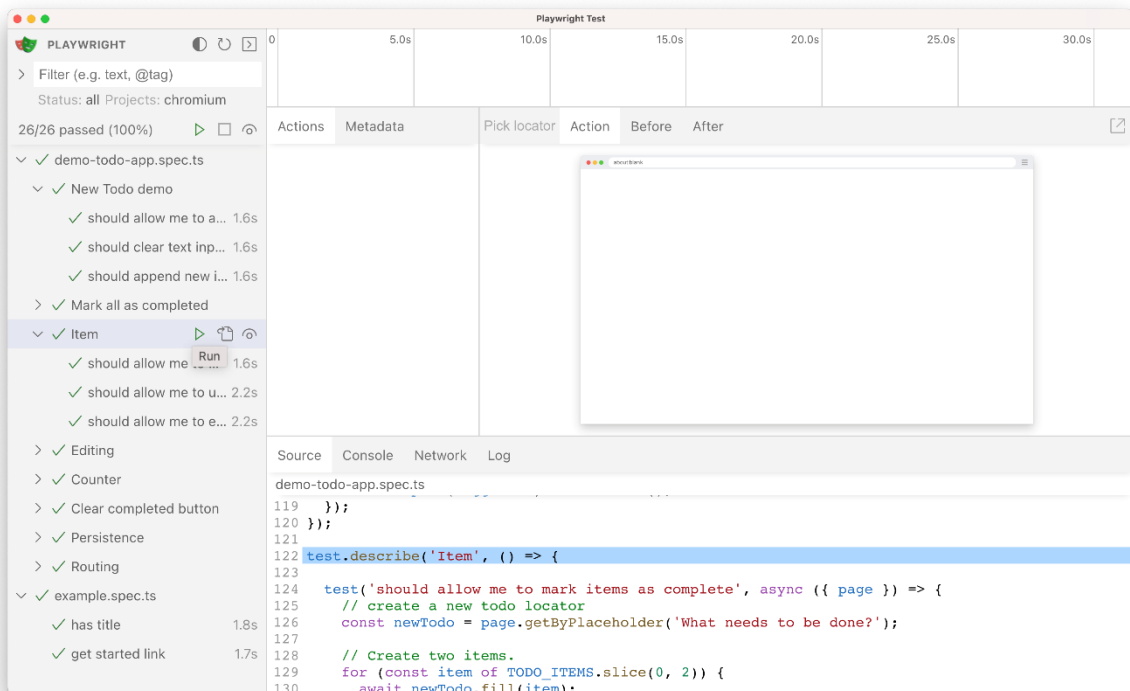
The Playwright Trace Viewer is also available online at <https://trace.playwright.dev>. Just drag-and-drop your “trace.zip” file to inspect its contents. In the online version, trace files are not uploaded anywhere; it is a progressive web application that processes traces locally.

UI Mode

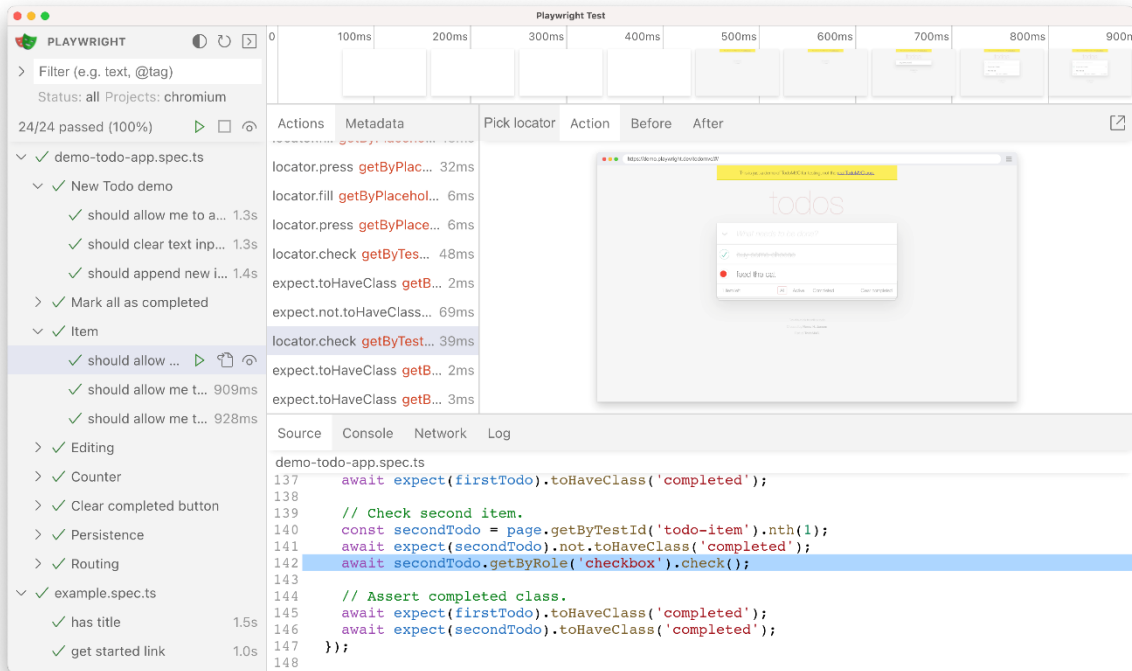
To open UI mode, run the following command:

```
npx playwright test --ui
```

Once you launch UI Mode you will see a list of all your test files. You can run all your tests by clicking the triangle icon in the sidebar. You can also run a single test file, a block of tests or a single test by hovering over the name and clicking on the triangle next to it.



Traces are shown for each test that has been run, so to see the trace, click on one of the test names. Note that you won't see any trace results if you click on the name of the test file or the name of a describe block.



In the Actions tab you can see what locator was used for every action and how long each one took to run. Hover over each action of your test and visually see the change in the DOM snapshot. Go back and forward in time and click an action to inspect and debug. Use the Before and After tabs to visually see what happened before and after the action. Next to the Actions tab you will find the Metadata tab which will show you more information on your test such as the Browser, viewport size, test duration and more.

As you hover over each action of your test the source code for the test is highlighted below. Click on the source tab to see the source code for the entire test. Click on the console tab to see the console logs for each action. Click on the log tab to see the logs for each action. Click on the network tab to see the network logs for each action.

Pop out the DOM snapshot into its own window for a better debugging experience by clicking on the pop out icon above the DOM snapshot (image). From there you can open the browser DevTools and inspect the HTML, CSS, Console etc. Go back to UI Mode and click on another action and pop that one out to easily compare the two side by side or debug each individually.

At the top of the trace you can see a timeline view of each action of your test. Hover back and forth to see an image snapshot for each action.

Click on the pick locator button and hover over the DOM snapshot to see the locator for each element highlighted as you hover. Click on an element to save the locator into the pick locator field. You can then copy the locator and paste it into your test.

Next to the name of each test in the sidebar you will find an eye icon. Clicking on the icon will activate watch mode, which will re-run the test when you make changes to it. You can watch a number of tests at the same time by clicking the eye icon next to each one or all tests by clicking the eye icon at the top of the sidebar. If you are using the Microsoft Visual Studio Code IDE, then you can easily open your test by clicking on the file icon ("Open in VS Code") next to the eye icon. This will open your test in Microsoft Visual Studio Code right at the line of code that you clicked on.

JavaScript Test Hooks and Fixtures

“test.beforeAll” and “test.afterAll” hooks are used to set up and tear down resources shared between tests. “test.beforeEach” and “test.afterEach” hooks are used to set up and tear down resources for each test individually.

```
const { test, expect } = require('@playwright/test');

test.describe('feature foo', () => {
  test.beforeEach(async ({ page }) => {
    // Go to the starting url before each test.
    await page.goto('https://playwright.dev/');
  });

  test('my test', async ({ page }) => {
    // Assertions use the expect API.
    await expect(page).toHaveURL('https://playwright.dev/');
  });
});
```

The argument “page” in this example code is called a “fixture” in Playwright. Here are the most commonly used Playwright fixtures:

Fixture	Type	Description
page	Page	Isolated page for this test run
context	BrowserContext	Isolated context for this test run. The page fixture belongs to this context as well.
browser	Browser	Browsers are shared across tests to optimize resources
browserName	String	The name of the browser currently running the test. Either chromium, firefox or webkit.

JavaScript Command line patterns

Run all the tests	<code>npx playwright test</code>
Run a single test file	<code>npx playwright test tests/todo-page.spec.ts</code>
Run a set of test files	<code>npx playwright test tests/todo-page/ tests/landing-page/</code>
Run files that have my-spec or my-spec-2 in the file name	<code>npx playwright test my-spec my-spec-2</code>
Run the test with the title	<code>npx playwright test -g "add a todo item"</code>
Run tests in headed browsers	<code>npx playwright test --headed</code>
Run tests on all browsers (chromium, firefox, webkit)	<code>npx playwright test --browser all</code>
Run tests in a particular configuration (project)	<code>npx playwright test --project=firefox</code>
Run all tests a number of times (for example 10 times to detect flakiness)	<code>npx playwright test --repeat-each 10</code>
Disable parallelization by running with a single worker	<code>npx playwright test --workers=1</code>
Choose a reporter	<code>npx playwright test --reporter=dot</code>
Run in debug mode with Playwright Inspector	<code>npx playwright test -debug</code>
Ask for help	<code>npx playwright test -help</code>

Core concepts of Playwright

Playwright cascades down over multiple levels. A browser is the highest level, followed by browser context(s), followed by pages and frames, which contain elements that are accessed by selectors. Selectors can also be repackaged as locators to make the program code easier to read and understand.

Browser

A browser refers to an instance of Chromium, Firefox, or WebKit. Playwright scripts generally start with launching a browser instance and end with closing the browser. Browser instances can be launched in headless (without a GUI), or headed mode.

Browser contexts

A browser context is an isolated incognito-alike session within a browser instance. Browser contexts are fast and cheap to create. Each test scenario should run in its own new browser context, so that the browser state is isolated between the tests.

Browser contexts can also be used to emulate multi-page scenarios involving mobile devices, permissions, locale and color scheme.

Pages and frames

A browser context can have multiple pages. A page refers to a single tab or a popup window within a browser context. A page should be used to navigate to URL's and interact with the page content.

A page can have one or more frame objects attached to it. Each page has a main frame and page-level interactions (like clicks) are assumed to operate in the main frame.

A page can have additional frames attached with the `iframe` HTML tag. These frames can be accessed for interactions inside the frame.

Selectors

Playwright can search for elements using CSS selectors, XPath selectors, HTML attributes like `id`, `data-test-id`, and even text content.

You can explicitly specify the selector engine you are using, or let Playwright detect it.

All selector engines except for XPath pierce shadow DOM by default. First they search for the elements in the light DOM in the iteration order, and then they search recursively inside open shadow roots in the iteration order.

In particular, in the CSS engine, any descendant combinator or child combinator pierces an arbitrary number of open shadow roots, including the implicit descendant combinator at the start of the selector. Playwright does not search inside closed shadow roots or iframes.

If you want to opt-out of this behaviour, then you can use the “:light” CSS extension or the “text:light” selector engine. They do not pierce shadow roots. You don’t typically need to do this, though.

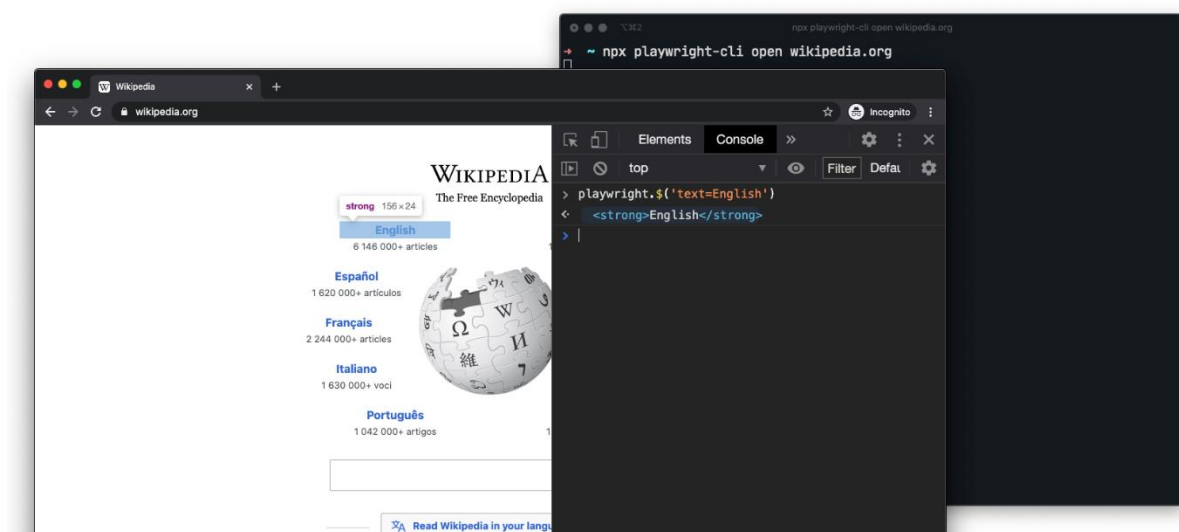
Some selector examples (there are more available, including special selectors for the React and Vue web frameworks):

Selector	Example
Find node by text substring	<code>page.click("text=Hello");</code>
Playwright supports a shorthand for selecting elements using these attributes: "id", "data-testid", "data-test-id", "data-test"	<code>page.click("data-test-id=foo");</code>
CSS and XPath selector engines are automatically detected (“xpath=” does not have to be prefixed, if the XPath starts with “//”)	<code>page.click("div");</code> <code>page.click("//html/body/div");</code>
Select by attribute, with CSS selector	<code>page.click("[aria-label=' Sign in ']");</code>
Selecting based on layout, with CSS selector	<code>page.click("input:right-of(:text(\"Username\"))");</code>
Pick n-th match. Note that unlike CSS's nth-match, the provided index is 0-based.	<code>page.click(":nth-match(:text(' Buy '), 3)");</code>
Only search light DOM, outside WebComponent shadow DOM	<code>page.click("css:light=div");</code>

Inspect Selectors

During “open” or “codegen”, you can use the following API inside the developer tools console of any browser.

Example: In Chromium, you open the Developer Tools with the “<F12>” key, and then click on “Console” to type:



Query Playwright selector, using the actual Playwright query engine:

```
playwright.$(selector) #
```

Same as “`playwright.$`”, but returns all matching elements:

```
playwright.$$ (selector) #
```

Reveal element in the elements panel (if the browser development tools of the respective browser supports it):

```
playwright.inspect (selector) #
```

Generates selector for the given element:

```
playwright.selector (element) #
```

Locators

Locators represent a view to the element(s) on the page. They capture the logic to retrieve elements at any given moment. This example looks for a link with the name “*Log in*” to click on:

```
Locator locator = page.getByRole("link", { name: "Log in" });  
locator.click();
```

The difference between a “Locator” and an “ElementHandle” is that the latter points to a particular element, while “Locator” captures the logic of how to retrieve that element.

Locators are particularly useful when using the Page Object Model design pattern.

Locators can make scripts much easier to read and understand.

Built-in locators

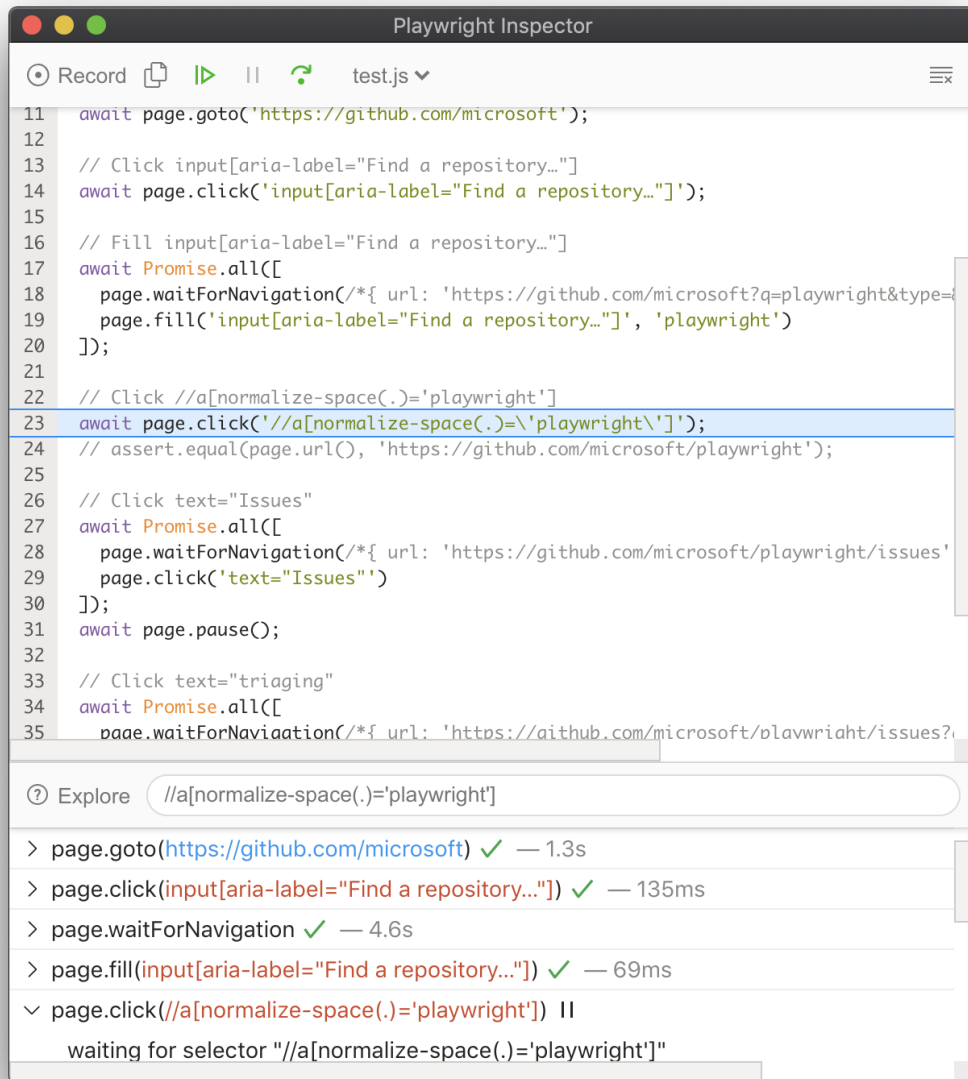
These are the recommended built in locators (“`getByRole`” should probably be the default choice):

Locator	To locate
<code>page.getByRole()</code>	explicit and implicit accessibility attributes
<code>page.getByText()</code>	text content
<code>page.getByLabel()</code>	a form control by associated label's text
<code>page.getByPlaceholder()</code>	an input by placeholder
<code>page.getByAltText()</code>	an element, usually image, by its text alternative
<code>page.getByTitle()</code>	an element by its title attribute
<code>page.getByTestId()</code>	an element based on its data-testid attribute (other attributes can be configured)

Playwright Inspector

There are many ways (such as the built-in browser developer tools) to debug Playwright scripts, but the Playwright Inspector is the default recommendation for script debugging.

The Playwright Inspector shows a toolbar to step through the Playwright script for debugging:



Open the Playwright Inspector

The Playwright Inspector can easily be used in the Command Line Interface (that is also used to record scripts).

The Playwright Inspector can also be called from within a script:

```
page.pause();
```

Java and JavaScript Command Line Tools

The Command Line Interface (CLI) offers many great features. The following examples are for Java, but the same Tools can also be used from JavaScript, for example to save and restore a state (as described in the paragraph “Preserve and restore authenticated state from command line”):

```
npx playwright open --save-storage=auth.json www.github.com
```

```
npx playwright open --load-storage=auth.json www.github.com
```

Record scripts automatically

Playwright can record scripts in Java. The Command Line Interface (CLI) can be used to record user interactions and generate Java code. This example records on the Google web site:

```
mvn  
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args="codegen google.com "
```

Preserve and restore authenticated state from command line

Run “codegen” with “--save-storage” to save cookies and localStorage at the end after recording. This is useful to separately record an authentication step and reuse it later. In this example, “auth.json” will contain the stored state:

```
mvn  
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args="codegen --save-storage=auth.json"
```

Run with “--load-storage” to restore a previously stored state. This way, all cookies and localStorage will be restored, bringing most web apps to the authenticated state.

Example of opening with restored state:

```
mvn  
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args="open --load-storage=auth.json my.web.app"
```

Example of recording with restored state.

```
mvn  
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args="codegen --load-storage=auth.json my.web.app"
```

Install browsers

Install all supported browsers:

```
mvn  
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args="install"
```

Open a page with browsers and emulation options

Open Chromium:

```
mvn
```

```
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args="open google.com"
```

Open WebKit:

```
mvn
```

```
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args="wk google.com"
```

Open emulate an Apple iPhone 13.

```
mvn
```

```
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args='open --device="iPhone 13" google.com'
```

Open emulate colour scheme and viewport (screen) size

```
mvn
```

```
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args="open --viewport-size=800,600 --color-scheme=dark twitter.com"
```

Open emulate geolocation, language and timezone

```
mvn
```

```
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args='open --timezone="Europe/Rome" --geolocation="41.890221,12.492348" --lang="it-IT" maps.google.com'
```

Take Screenshot

Wait 3 seconds before capturing a screenshot after page loads (“load” event fires):

```
mvn
```

```
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args='screenshot --device="iPhone 13" --color-scheme=dark --wait-for-timeout=3000 twitter.com twitter-iphone.png'
```

Capture a full page screenshot:

```
mvn
```

```
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args='screenshot --full-page google.com google-full.png'
```

Generate PDF

PDF generation only works in headless Chromium:

```
mvn
```

```
exec:java -e -Dexec.mainClass=com.microsoft.playwright.CLI -Dexec.args="pdf https://google.com google.pdf"
```

Auto Waiting

Playwright automatically performs these actionability checks:

Action	Attached	Visible	Stable	Receives Events	Enabled	Editable
check	Yes	Yes	Yes	Yes	Yes	-
click	Yes	Yes	Yes	Yes	Yes	-
dblclick	Yes	Yes	Yes	Yes	Yes	-
Tap	Yes	Yes	Yes	Yes	Yes	-
uncheck	Yes	Yes	Yes	Yes	Yes	-
hover	Yes	Yes	Yes	Yes	-	-
scrollIntoViewIfNeeded	Yes	Yes	Yes	-	-	-
screenshot	Yes	Yes	Yes	-	-	-
fill	Yes	Yes	-	-	Yes	Yes
selectText	Yes	Yes	-	-	-	-
dispatchEvent	Yes	-	-	-	-	-
focus	Yes	-	-	-	-	-
getAttribute	Yes	-	-	-	-	-
innerText	Yes	-	-	-	-	-
innerHTML	Yes	-	-	-	-	-
press	Yes	-	-	-	-	-
setInputFiles	Yes	-	-	-	-	-
selectOption	Yes	-	-	-	-	-
textContent	Yes	-	-	-	-	-
type	Yes	-	-	-	-	-

It is possible to wait for a specific selector:

```
page.waitForSelector("text=My Text") ;
```

Although Auto Waiting will usually do the work, it is also possible to check programmatically:

```
ElementHandle.isChecked()  
ElementHandle.isDisabled()  
ElementHandle.isEditable()  
ElementHandle.isEnabled()  
ElementHandle.isHidden()  
ElementHandle.isVisible()  
Page.isChecked(selector[, options])  
Page.isDisabled(selector[, options])  
Page.isEditable(selector[, options])
```

```
Page.isEnabled(selector[, options])
```

```
Page.isHidden(selector[, options])
```

```
Page.isVisible(selector[, options])
```

Inputs

Text input (“fill”)

This is the easiest way to fill in form fields. It focuses on the element and triggers an input event with the entered text. It works for “<input>”, “<textarea>”, “[contenteditable]” and “<label>” associated with an input or textarea. For example:

```
page.fill("text=First Name", "Peter");
```

Checkboxes and radio buttons (“check” and “uncheck”)

This is the easiest way to check and uncheck a checkbox or a radio button. This method can be used with “input[type=checkbox]”, “input[type=radio]”, “[role=checkbox]” or “label” associated with checkbox or radio button. Examples:

```
page.check("#agree");
```

```
page.uncheck("#agree");
```

Select options (“selectOption”)

Selects one or multiple options in the “<select>” element. You can specify the option value, label or elementHandle to select. Multiple options can be selected. Examples:

```
page.selectOption("select#colors", "blue");
```

```
page.selectOption("select#colors", new String[] {"red", "green", "blue"});
```

Mouse click (“click” and “dblclick”)

```
page.click("button#submit");
```

```
page.dblclick("#item");
```

Type characters (“type”)

This method will emit all the necessary keyboard events, with all the “keydown”, “keyup”, “keypress” events in place. You can even specify the optional delay between the key presses to simulate real user behaviour. For example:

```
page.type("#area", "Hello World!");
```

Keys and shortcuts (“press”)

This method focuses the selected element and produces a single keystroke. It accepts the logical key names that are emitted in the “`keyboardEvent.key`” property of the keyboard events like:

Backquote, Minus, Equal, Backslash, Backspace, Tab, Delete, Escape, ArrowDown, End, Enter, Home, Insert, PageDown, PageUp, ArrowRight, ArrowUp, F1 - F12, Digit0 - Digit9, KeyA - KeyZ, etc.

Example:

```
page.press("#name", "Shift+A");
```

Java and JavaScript Assertions

Playwright provides convenience API’s for common tasks, like reading the text content of an element. These API’s can be used in your test assertions.

Java Asserption examples

Assertion	Example
Text content	<pre>String <i>content</i> = page.textContent("nav:first-child"); assertEquals("home", <i>content</i>);</pre>
Inner text	<pre>String <i>text</i> = page.innerText(".selected"); assertEquals("value", <i>text</i>);</pre>
Attribute value	<pre>String <i>alt</i> = page.getAttribute("input", "alt"); assertEquals("Text", <i>alt</i>);</pre>
Checkbox state	<pre>boolean <i>checked</i> = page.isChecked("input"); assertTrue(<i>checked</i>);</pre>
JS expression	<pre>Object <i>content</i> = page.evalOnSelector("nav:first-child", "e => e.textContent"); assertEquals("home", <i>content</i>);</pre>
Inner HTML	<pre>String <i>html</i> = page.innerHTML("div.result"); assertEquals("<p>Result</p>", <i>html</i>);</pre>
Visibility	<pre>boolean <i>visible</i> = page.isVisible("input"); assertTrue(<i>visible</i>);</pre>
Enabled state	<pre>boolean <i>enabled</i> = page.isEnabled("input"); assertTrue(<i>enabled</i>);</pre>

JavaScript Assertion examples (just some of the same Assertions as in Java)

Assertion	Example
Text content	<pre>const <i>content</i> = await page.textContent('nav:first-child'); expect(<i>content</i>).toBe('home');</pre>
Attribute value	<pre>const <i>alt</i> = await page.getAttribute('input', 'alt'); expect(<i>alt</i>).toBe('Text');</pre>
Checkbox state	<pre>const <i>checked</i> = await page.isChecked('input'); expect(<i>checked</i>).toBeTruthy();</pre>
Visibility	<pre>const <i>visible</i> = await page.isVisible('input'); expect(<i>visible</i>).toBeTruthy();</pre>

Java and JavaScript Reuse authentication states programmatically

Save the authentication state programmatically

The authentication state (cookies and local storage) can be saved to a file and later restored.

This is a very powerful feature that can save a lot of time and effort to reuse authentication states in apps that that require (login) authentication.

In Java:

```
context.storageState(new  
BrowserContext.StorageStateOptions().setPath(Paths.get("auth.json")));
```

In JavaScript:

```
await context.storageState({ path: 'auth.json' });
```

Create a new context with the saved storage state

In Java:

```
BrowserContext context = browser.newContext(new  
Browser.NewContextOptions().setStorageStatePath(Paths.get("auth.json")));
```

In JavaScript:

```
const context = await browser.newContext({ storageState: 'auth.json' });
```

Emulation

Playwright allows overriding various parameters of the device where the browser is running:

- User agent
- Viewport (viewport size, device scale factor, touch support)
- Locale & timezone
- Permissions (such as notifications and geolocation access)
- Geolocation
- Color scheme and media

Most of these parameters are configured during the browser context construction, but some of them (such as viewport size) can be changed for individual pages.

Browser Flags and configuration settings (Options)

In Playwright, browser settings can be set with this command:

```
BrowserType.launch([options])
```

Chromium Flags

Run Chromium with flags:

<http://www.chromium.org/developers/how-tos/run-chromium-with-flags>

List of Chromium command line switches:

<http://peter.sh/experiments/chromium-command-line-switches/>

List of default flags:

<https://github.com/christian-fei/mega-scrapers/blob/master/lib/browser/get-puppeteer-options.js>

Use this command in Chrome-based browsers to see the full list of what is enabled or disabled:

chrome://flags

Mozilla Firefox configuration settings

Use this command in the Mozilla Firefox browser to see configuration settings:

about:config

A list of all Mozilla Firefox “about” pages:

about:about

JavaScript Data-Driven Tests

This example checks the homepages of Google, Apple, and Microsoft.

1. Create a data file “urls.json” in a folder called “data” with this content:

```
[  
  "https://www.google.com",  
  "https://www.apple.com",  
  "https://www.microsoft.com"  
]
```

2. You can now use this data to drive a Data-Driven Test:

```
const { test, expect } = require('@playwright/test');  
const urls = require('../data/urls.json');  
for (const url of urls) {  
  test(`check ${url}`, async ({ page }) => {  
    await page.goto(url);  
  })  
}
```

JavaScript Tags

All test blocks (“describe”) and all tests inside the blocks (“test”) should be tagged.

This allows for selective execution of test blocks and/or tests with specific tags.

In Playwright, grep just filters for a matching regular expression. This means that you can filter for any text (substring) of the test name and/or tags. Tags are therefore just a part of the test name. The convention is to prefix tags with a “@” character.

Example use of tags

Tags can be used on test blocks (“describe”), and/or on tests (“test”), as in this example:

```
const { test, expect } = require('@playwright/test');

test.describe('block with a tag @regression', () => {

  test('example tag test one @firstTag @secondTag', () => {
    expect(true).toBe(true);
  });

  test('example tag test two @firstTag', () => {
    expect(true).toBe(true);
  });

});
```

Filtering with grep

Command line tag filtering examples:

```
# run all blocks or tests with the tag “@regression”
$ npx playwright test --grep '@regression'

# run all blocks or tests with the tag “@secondTag”
$ npx playwright test --grep '@secondTag'

# run all blocks or tests except those with the tag “@secondTag”
$ npx playwright test --grep-invert '@secondTag'

# run all blocks or tests with the text (substring) “test two” in the name
$ npx playwright test --grep 'test two'
```

Configuration file tag filtering example

You can also filter from the configuration file “playwright.config.js”, as in this example:

```
...
grep: [new RegExp("@secondTag")],
...
```

Playwright on Microsoft .net and Azure DevOps

This section describes the installation steps for Playwright on the Microsoft .net platform with or without the Behaviour-Driven Development (BDD) tool SpecFlow.

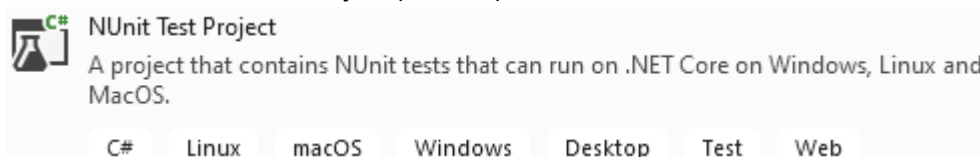
It also describes the Playwright test integration and test execution from Microsoft Azure DevOps.

Playwright installation without an IDE (NUnit)

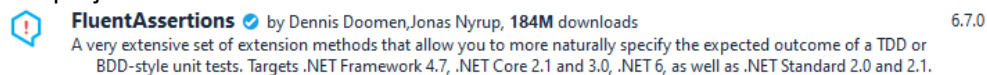
1. Initialize the project in a new project folder:
`dotnet new nunit -n <projectname>`
2. Change into the newly created project folder:
`cd <projectname>`
3. Install NUnit package:
`dotnet add package Microsoft.Playwright.NUnit`
4. Build the project:
`dotnet build`

Playwright installation in Visual Studio (NUnit without SpecFlow)

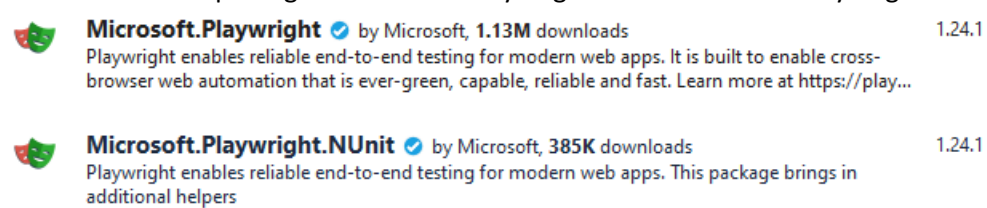
1. Create a new NUnit Test Project (.net 6.0):



2. Install the NuGet package “FluentAssertions”, if it has not already been added during the project creation:



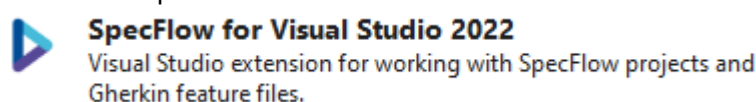
3. Install the NuGet packages “Microsoft.Playwright” and “Microsoft.Playwright.NUnit”:



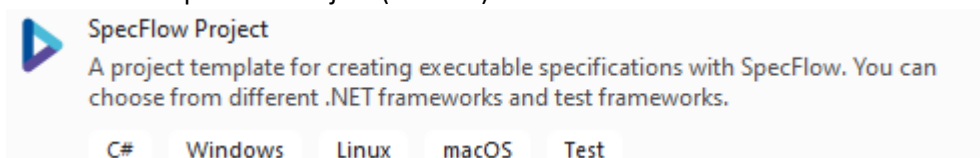
4. You can replace the existing “UnitTest1.cs” test with the Playwright example test from <https://playwright.dev/dotnet/docs/intro> .

Playwright installation in Visual Studio (NUnit with SpecFlow)



1. Install the “SpecFlow for Visual Studio 2022” Extension:





2. Create a new SpecFlow Project (.net 6.0):





3. Install the NuGet package “FluentAssertions”, if it has not already been added during the SpecFlow project creation:

 **FluentAssertions**  by Dennis Doomen,Jonas Nyrup, **184M** downloads 6.7.0
A very extensive set of extension methods that allow you to more naturally specify the expected outcome of a TDD or BDD-style unit tests. Targets .NET Framework 4.7, .NET Core 2.1 and 3.0, .NET 6, as well as .NET Standard 2.0 and 2.1.

4. Install the NuGet packages “Microsoft.Playwright” and “Microsoft.Playwright.NUnit”:

 **Microsoft.Playwright**  by Microsoft, **1.13M** downloads 1.24.1
Playwright enables reliable end-to-end testing for modern web apps. It is built to enable cross-browser web automation that is ever-green, capable, reliable and fast. Learn more at <https://playwright.dev>


 **Microsoft.Playwright.NUnit**  by Microsoft, **385K** downloads 1.24.1
Playwright enables reliable end-to-end testing for modern web apps. This package brings in additional helpers

5. Install the NuGet package “SpecFlow.Assist.Dynamic” (for easier data table handling):

 **SpecFlow.Assist.Dynamic** by www.marcusoft.net - Marcus Hammarberg, **2.31M** 1.4.2
See documentation at <https://github.com/marcusoftnet/SpecFlow.Assist.Dynamic/wiki/Documentation>

Additional installations for Azure DevOps

1. Install the “Azure DevOps Test” Extension:

 **Azure DevOps Test Connector (64-bit)**
A Visual Studio extension allowing users to link Test Classes and SpecFlow Feature files with Azure DevOps Test Plans/Suites and Cases using attributes. Now featuring support for Specflow Scenario Out

2. Configure the “Azure DevOps Test” Extension in Visual Studio:

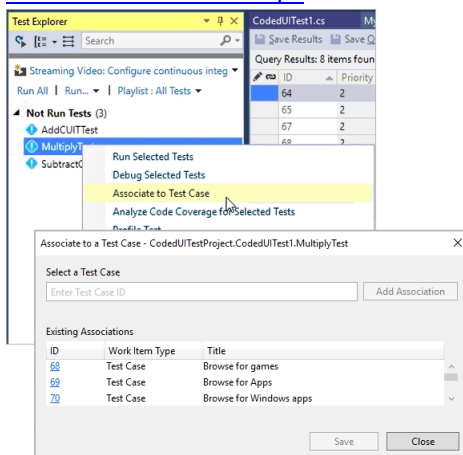
“Tools” -> “Options...” -> “Azure DevOps Test Connector”.

Set the following 3 settings:

1. “Azure DevOps instance URL” must be the base URL of the Azure DevOps instance you want to connect to, for example:
“<https://myinstance.visualstudio.com>”.
2. “Current Project name”, for example “*myprojectname*”.
3. “PAT Code” must be the Personal Access Token. Use the pure token itself, the user name must not be used. For example:
“2s23b43tsjeevgec7smc6acba5lcok7afua3otegeywc2wwj2fcq”.

3. Link the test cases to Azure DevOps using the Visual Studio Test Explorer:

<https://learn.microsoft.com/en-us/azure/devops/test/associate-automated-test-with-test-case?view=azure-devops>



Install Browsers on Microsoft .net (in PowerShell)

This is only required once per system. Please note that the net version number might differ:

```
pwsh bin\Debug\net6.0\playwright.ps1 install --with-deps
```

Running tests on Microsoft .net

```
dotnet test -- NUnit.NumberOfTestWorkers=5
```

Visual Studio “.runsettings” file

A file named exactly “.runsettings” can only be auto-detected by Visual Studio, if it is stored at the solution file level (where the “.sln” file is stored) level.

The following is an example of a “.runsettings” file:

- Defines 5 worker processes
- Defines a URL as a parameter (similar to “baseUrl”)
- Shows the browser during execution (headed mode)

```
<?xml version="1.0" encoding="utf-8"?>
<RunSettings>
  <NUnit>
    <NumberOfTestWorkers>5</NumberOfTestWorkers>
  </NUnit>
  <TestRunParameters>
    <Parameter name="PlaywrightHomepage" value="https://playwright.dev" />
  </TestRunParameters>
  <RunConfiguration>
    <EnvironmentVariables>
      <HEADED>1</HEADED>
    </EnvironmentVariables>
  </RunConfiguration>
</RunSettings>
```

Additional information:

<https://playwright.dev/dotnet/docs/next/test-runners#using-the-runsettings-file>